𝕋ℂ𝕊-𝕋ℝ-𝔸-10-41

𝕋ℂ𝕊 Technical Report

Mining Approximate Patterns with Frequent Locally Optimal Occurrences

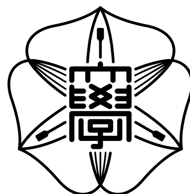
by

Atsuyoshi Nakamura
Hisashi Tosaka
Mineichi Kudo

Division of Computer Science

Report Series A

March 1, 2010



Hokkaido University
Graduate School of
Information Science and Technology

Email: atsu@ist.hokudai.ac.jp

Phone: +81-011-706-6806
Fax: +81-011-706-7832

Mining Approximate Patterns with Frequent Locally Optimal Occurrences

March 1, 2010

Abstract

We propose a novel frequent approximate pattern mining that suits estimation of occurrence regions. Given a string s, our mining enumerates its substrings that locally optimally match many substrings of s. We show an algorithm for this problem in which candidate patterns are generated without duplication using the suffix tree of s. This problem can be extended to the problem of enumerating approximate frequent subforests of a given ordered labeled tree T. Our mining was applied to the task of extraction of search result records from a web page returned by a search engine, and had good performance for benchmark data sets.

1 Introduction

Now, frequent mining is one of the most popular study areas in data mining. In the area of frequent mining, strings and trees are popular objects that people are interested in, and various problems on them have been studied so far in order to find useful patterns. Frequent contiguous exact patterns like substrings and bottom-up subtrees can be enumerated easily, but such restriction on patterns is too strict to find useful patterns in real data.

Two approaches are there to overcome this problem. One approach is to remove contiguous restriction, namely, to consider flexible matching. In this approach, subsequences for strings [1], induced [2] and embedded [12] subtrees for trees were studied in the context of frequent mining. The other approach is to remove exact restriction, namely, to consider approximate patterns. As for frequent substring mining in this approach, a frequent mining that counts the number of similar substrings with respect to Hamming distance was studied [15].

In this paper, we study frequent mining of the latter approach for strings and labeled ordered trees. The problem considered here is to find patterns frequently appearing in one string or tree instead of a set of strings or trees. Applications of such mining include impressive melody part extraction by finding frequent contiguous

subsequences (substrings) in a melody sequence and data record extraction by finding frequent subforests in a tag tree of a web page. In such applications, the region of each occurrence must be estimated because the purpose is not to find patterns but to extract occurrences correctly.

In frequent approximate mining from one given string or tree, previous Hamming distance approach is not sufficient because one insertion or deletion changes Hamming distance drastically while it changes edit distance a little. However, edit distance approach makes a counting problem arise: essentially the same part may be counted many times because a small change in range boundaries does not produce a big difference in edit distance. This is not a problem when frequency is counted by how many strings contain occurrences of a pattern in a given set of strings. But, this is a big problem when every occurrence in a string is counted. Furthermore, we are considering a problem of extracting occurrences correctly, which means that occurrence boundaries must be estimated appropriately.

To overcome this problem, for each pattern, we count its *locally optimal occurrences* in a given string, where a substring $s[h..j]$ of $s[1..n]$ is said to be a locally optimal occurrence of pattern $p[1..m]$ if the pair of $s[h..j]$ and $p[1..m]$ is *locally optimal* [6] as substrings of $s[1..n]$ and $p[1..m]$. Here, $s[h..j]$ denotes the substring of s from position h to position j. A locally optimal occurrence of pattern p has optimized boundaries in the meaning that the boundaries are best fitted to p in terms of alignment score when one of the boundaries is fixed. By counting locally optimal occurrences alone, only the optimal one among occurrences with slightly different boundaries is counted and such an optimal occurrence is considered to be appropriate as an estimated occurrence region of a pattern. By restricting patterns to substrings in a given sting, and by generating all substring without duplication using the suffix tree of the string, approximate patterns with frequent locally optimal occurrences can be enumerated in $O(n^3)$ time and $O(n^3)$ space for a given string with length n. All locally optimal pairs are known to be calculated by both forward and backward executions of a local alignment algorithm [9], and our algorithm is a harmonic combination of the algorithm and pattern generation using a suffix tree.

Since a labeled ordered tree can be regarded as a layered strings by seeing the sequence of child subtrees for each internal node as a string, the problem and the algorithm for a string can be extended to those for a labeled ordered tree, where child subtrees for an internal node d are subtrees rooted by child nodes of d. As local optimality, we add one more condition, *upward-downward optimality*, which requests that an occurrence subforest has the best score among all the subforests contained in it or containing it. We show $O(n^4)$-time $O(n^3)$-space algorithm for enumerating subforest patterns in a given tree with n nodes. Compared to the algorithm for a string, its time complexity is $O(n)$ factor larger, which is the time needed to check upward-downward optimality. $O(n^4)$-time seems too slow for practical use, but it took about $O(m^{1.73})$ only according to our experiments on search result record extraction, where m is the total length of distinct child subtree sequences for all internal nodes,

which is at most n.

As an application of our algorithm for a labeled ordered tree, we propose a method for search result record (SRR) extraction based on mining approximate subforest patterns with frequent locally optimal occurrences. One merit of using frequent mining for this problem is to be able to extract SRRs even if any pair of SRRs does not appear contiguously, which is impossible for popular methods [8, 10, 14] that detect contiguous occurrences to find SRRs. This merit could be demonstrated in our experiments using ViNTs data sets. According to our experimental comparison with ViNTs [14], which is a state-of-the-art system for SRR extraction, our method performed between the two versions of ViNTs, a version using visual features and a version using no such feature, with respect to F-measure, namely, the harmonic average of precision and recall. Considering performance improvement potential of our method by using visual features, this result demonstrates effectiveness of our method as a base method of SRR extraction.

2 Mining Approximate Strings

2.1 Problem Setting

Let Σ be a finite *alphabet*, whose elements are called *letters*. A *string* s is a sequence of letters with finite length. The length of string s is the number of letters in s which is denoted by $|s|$. We let $s[i]$ denote the ith letter of s, so s is also expressed by $s[1]s[2]\cdots s[n]$ when $|s| = n$. We also use notation $s[i..j]$ for $j \geq i-1$ which is equal to $s[i]s[i+1]\cdots s[j]$, namely, a substring of s that begins at the ith letter and ends at the jth letter. Note that $s[i..i-1]$ denotes a *null (empty) string*, which is a substring of any string. Let '-' denote the *gap symbol* which is assumed not to belong to Σ. A pair (s'_1, s'_2) of strings is said to be a *(global) alignment* of strings s_1 and s_2 if the following three conditions are satisfied: (1) s'_1 and s'_2 are made from s_1 and s_2, respectively, by inserting gap symbols, (2) $|s'_1| = |s'_2|$, and (3) $s'_1[i] = s'_2[i] =$'-' for no $i \in \{1, 2, ..., |s'_1|\}$.A *score function* $w(x, y)$ is a real-valued function on $(\Sigma \cup \{\text{-}\}) \times (\Sigma \cup \{\text{-}\})$ that have the following properties: $w(x, y) = w(y, x)$ for all $x, y \in \Sigma \cup \{\text{-}\}$ and $w(x, \text{-}) < 0 < w(x, x)$ for all $x \in \Sigma$. For an alignment (s'_1, s'_2) of strings s_1 and s_2, its score $S_{\mathrm{AL}}(s'_1, s'_2)$ is defined as

$$S_{\mathrm{AL}}(s'_1, s'_2) = \sum_{i=1}^{|s'_1|} w(s'_1[i], s'_2[i])$$

Let $\mathrm{AL}(s_1, s_2)$ denote the set of all alignments of strings s_1 and s_2. An alignment $(s^*_1, s^*_2) \in \mathrm{AL}(s_1, s_2)$ is said to be *optimal* if

$$S_{\mathrm{AL}}(s^*_1, s^*_2) = \max_{(s'_1, s'_2) \in \mathrm{AL}(s_1, s_2)} S_{\mathrm{AL}}(s'_1, s'_2).$$

We let $S_{\mathrm{OPT}}(s_1, s_2)$ denote $\max_{(s'_1, s'_2) \in \mathrm{AL}(s_1, s_2)} S_{\mathrm{AL}}(s'_1, s'_2)$.

We define an approximate occurrence of a pattern p with locally optimal boundaries in terms of alignment score.

Definition 1 *A substring* $s[h..j]$ *of* $s[1..n]$ *is said to be a* locally optimal occurrence *of string* $p[1..m]$ *if the following conditions are satisfied:*

CS1 $S_{OPT}(p[1..m], s[h..j]) > 0$,

CS2 $S_{OPT}(p[1..m], s[h..j]) \geq S_{OPT}(p[g..m], s[h'..j])$ *for all* $1 \leq g \leq m+1, 1 \leq h' \leq j+1$, *and*

CS3 $S_{OPT}(p[1..m], s[h..j]) \geq S_{OPT}(p[1..m'], s[h..j'])$ *for all* $0 \leq m' \leq m, h-1 \leq j' \leq n$.

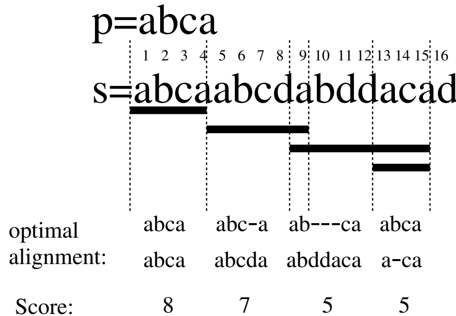
Condition CS1 says that $s[h..j]$ must be similar to $p[1..m]$, and condition CS2(CS3) says that the left(right) end of $s[h..j]$ is optimal when the right(left) ends of both $s[h..j]$ and $p[1..m]$ are fixed. Note that the definition above can be also expressed using a notion of a *locally optimal pair* introduced by Erickson and Sellers [6]: a substring $s[h..j]$ of $s[1..n]$ is said to be a *locally optimal occurrence* if a pair of $s[h..j]$ and $p[1..m]$ is locally optimal as substrings of $s[1..n]$ and $p[1..m]$. The purpose of introducing a new notion is to distinguish a pattern from an occurrence.

Example 1 *Let p ="abca" and s="abcaabcdabddacad". Define score function w as follows:*

$$w(x,y) = \begin{cases} 2 & (x = y,\ x, y \in \Sigma) \\ -2 & (x \neq y,\ x, y \in \Sigma) \\ -1 & (x = \text{-} \ \ or \ \ y = \text{-}) \end{cases}$$

Then, locally optimal occurrences of p in s are the following four substrings of s: (See the left figure.)

$$\begin{array}{lcll} s[1..4] & = & \text{“abca”} & (S_{OPT}(p, s[1..4]) = 8), \\ s[5..9] & = & \text{“abcda”} & (S_{OPT}(p, s[5..9]) = 7), \\ s[9..15] & = & \text{“abddaca”} & (S_{OPT}(p, s[9..15]) = 5), \\ s[13..15] & = & \text{“abca”} & (S_{OPT}(p, s[13..15]) = 5). \end{array}$$



Note that String $s[9..13]$ *="abdda" is not a locally optimal occurrence of p even if* $s[14..15] \neq$ *"ca" because* $3 = S_{OPT}(p, s[9..13]) < S_{OPT}(p[1..2], s[9..10]) = 4$, *which violates CS3.*

As you can see from the above example, a locally optimal occurrence matches to the pattern very well near boundaries, which reduces the number of occurrences for essentially the same parts comparing to the simple method counting all the substrings whose edit distance from the pattern is at most a given threshold. Boundary fitness also seems desirable for region estimation.

Definition 2 *A substring $s[h..j]$ of $s[1..n]$ is said to be a* minimal *locally optimal occurrence of string $p[1..m]$ if $s[h..j]$ is a locally optimal occurrence of $p[1..m]$ and satisfies the following minimality conditions:*

CS2M $S_{OPT}(p[1..m], s[h..j]) > S_{OPT}(p[g..m], s[h'..j])$ *for all* $1 \leq g \leq m+1, h \leq h' \leq j+1$ *but* $(g, h') = (1, h)$, *and*

CS3M $S_{OPT}(p[1..m], s[h..j]) > S_{OPT}(p[1..m'], s[h..j'])$ *for all* $0 \leq m' \leq m, h-1 \leq j' \leq j$ *but* $(m', j') = (m, j)$.

Note that there might be more than one locally optimal occurrences $s[h..j]$ for each h (for each j) but the minimal one $s[h..j_0]$ ($s[h_0..j]$) is unique.

For computational efficiency, pattern strings are limited to substrings in a given string s in this paper. We consider the following frequent mining problem.

Problem 1 *Given a string s, a score function w and a natural number σ, enumerate distinct substrings $s[h..j]$ such that the number of minimal locally optimal occurrences of $s[h..j]$ in s is at least σ.*

Example 2 *Consider s and w defined in Example 1. Then, the number of minimal locally optimal occurrences of $s[1..4]$ = "abca" is three because $s[9..15]$ = "abddaca" only is not minimal among its four locally optimal occurrences. Note that the set of occurrences becomes disjoint in this case by virtue of minimality introduction. When $\sigma = 3$, the solution of the mining problem defined above is "a", "b", "c", "d", "ab", "ac", "ca", "abc", "aca", "cda", "abca", "abcda". Notice that frequent substrings do not have anti-monotone property in this problem.*

2.2 Algorithm

Given a pattern string $p[1..m]$, the number of minimal locally optimal occurrences of $p[1..m]$ in $s[1..n]$ can be obtained utilizing an algorithm for the local alignment problem. Actually, all locally optimal pairs are known to be enumerated by running such an algorithm in both backward and forward directions [9]. Here, we propose an algorithm into which this technique is incorporated together with efficient generation of patterns without duplication using a suffix tree.

```
EnumSubstrFLOO(s)
Input:    s[1..n]: string

1: Make the suffix tree T of string s.
2: for j = 0 to n do (D[j], H[j]) ← (0, (j + 1, 1))
3: v ←the root node of T
4: Optimality_Check(s, v, 0, D, H,'F')
5: Make the suffix tree T^r of the reversed string s^r.
6: for j = 0 to n do (D[j], H[j]) ← (0, (j + 1, 1))
7: v^r ←the root node of T^r
8: Optimality_Check(s^r, v^r, 0, D, H,'B')

Optimality_Check(s, v, l, D_0, H_0,mode)
Input:    s[1..n]: (reversed) string
          v: a node of the suffix tree of s
          l: the depth of node v
          D_0: D_0[j] = D[l, j]
          H_0: H_0[j] = H[l, j]

1: for each child node u of node v do
2:   (f, t) ← label of edge (v, u)
3:   for j = 0 to n do (D[j], H[j]) ← (D_0[j], H_0[j])
4:   for j = f to t do
5:     if mode=F then
6:       Forward_Check(s, s[f − l..j], D, H)
7:     else
8:       occ_list← list stored at the position j of edge (v, u)
9:       if Backward_Check(s, s[f − l..j], D, H, occ_list) ≥ σ then
10:          Print s[n−j+1..n−(f−l)+1] as one of the answers
11:       end if
12:     end if
13:   end for
14:   EnumFreqPat(s, u, l + t − f + 1, D, H,mode)
15: end for
```

Figure 1: Algorithms EnumSubstrFLOO and Optimality_Check

Let $D[i,j]$ be the maximum score among the scores $S_{\mathrm{OPT}}(p[g..i], s[h..j])$ for all $1 \le g \le i+1, 1 \le h \le j+1$. This value is known to be calculated by dynamic programming based on the following recursive formula:

$$D[i,j] \leftarrow \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ \text{and otherwise,} & \\ \max \begin{cases} 0 \\ D[i-1,j] + w(p[i], \text{-}) \\ D[i,j-1] + w(\text{-}, s[j]) \\ D[i-1,j-1] + w(p[i], s[j]). \end{cases} \end{cases}$$

From the definition of $D[i,j]$, condition CS2 is satisfied for $s[h..j]$ if and only if $S_{\mathrm{OPT}}(p[1..m], s[h..j]) = D[m,j]$.

Let $H[i,j]$ be the largest pair (h,g) of starting indexes in lexicographical order among those (h,g) that satisfies $S_{\mathrm{OPT}}(p[g..i], s[h..j]) = D[i,j]$:

$$H[i,j] = \max\{(h,g) : 1 \le h \le j+1, 1 \le g \le i+1, S_{\mathrm{OPT}}(p[g..i], s[h..j]) = D[i,j]\}.$$

The definition of $H[i,j]$ implies that conditions CS2 and CS2M are satisfied for $s[h..j]$ if and only if $H[m,j] = (h,1)$.

Forward_Check(s,p,D,H)
Input: $s[1..n]$: string, $p[1..m]$: pattern
Update: D: $D[j] = D[m-1,j] \to D[m,j]$
H: $H[j] = H[m-1,j] \to H[m,j]$

1: CalDH(s,p[m],D,H)
2: occ_list←NULL
3: **for** $j = 1$ to n **do**
4: $(h,g) \leftarrow H[j]$
5: **if** $D[j] > 0$ and $g = 1$ **then**
6: Add (h,j) to the head of occ_list
7: **end if**
8: **end for**
9: Store occ_list at an appropriate position of the edge corresponding to p^r in the suffix tree $\mathcal{T}^r$

Backward_Check(s^r, p^r, D, H,occ_list)
Input: $s^r[1..n]$: reversed string, $p^r[1..m]$: reversed pattern
Update: D: $D[j] = D[m-1,j] \to D[m,j]$
H: $H[j] = H[m-1,j] \to H[m,j]$
Output: f: number of minimal locally optimal occurrences of p

1: CalDH($s^r, p^r[m], D, H$)
2: **if** occ_list= NULL **then** return 0
3: $f \leftarrow 0$
4: $(h_F, j_F) \leftarrow$ the first element of occ_list
5: **for** $j = 1$ to n **do**
6: **if** $j = n - h_F + 1$ **then**
7: $(h,g) \leftarrow H[j]$
8: **if** $D[j] > 0$, $g = 1$ and $h = n - j_F + 1$ **then**
9: $f \leftarrow f + 1$
10: **end if**
11: **if** no next element in occ_list **then** return f
12: $(h_F, j_F) \leftarrow$ the next element of occ_list
13: **end if**
14: **end for**

CalDH(s,c,D,H)
Input: $s[1..n]$: string
c: the last letter $p[m]$ of pattern $p[1..m]$
Update: D: $D[j] = D[m-1,j] \to D[m,j]$
H: $H[j] = H[m-1,j] \to H[m,j]$

1: $(d^*, (h^*, g^*)) \leftarrow (0, (1, 2))$
2: **for** $j = 1$ to n **do**
3: $(d_{-1}, (h_{-1}, g_{-1})) \leftarrow (d^*, (h^*, g^*))$
4: $(d^*, (h^*, g^*)) \leftarrow \max\{(0, (j+1, 2)), (D[j] + w(c,\text{-}), H[j]), (d_{-1} + w(\text{-}, s[j]), (h_{-1}, g_{-1})), (D[j-1] + w(c, s[j]), H[j-1])\}$
5: $(D[j-1], H[j-1]) \leftarrow (d_{-1}, (h_{-1}, g_{-1}))$
6: **end for**
7: $(D[n], H[n]) \leftarrow (d^*, (h^*, g^*))$

Figure 2: Algorithms Forward_Check, Backward_Check and CalDH

Note that $H[i,j]$ can be also calculated using the following recursive formula:

$$H[i,j] \leftarrow \begin{cases} (j+1, 1) & \text{if } i = 0 \\ \max J[i,j] & \text{otherwise,} \end{cases}$$

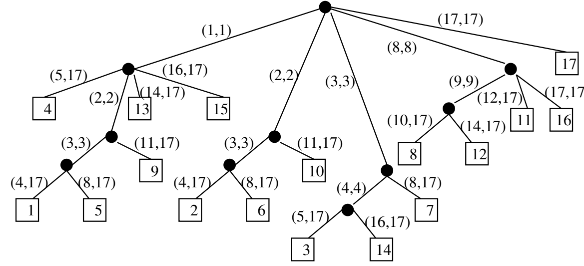
where

$$\begin{aligned} & J[i,j] \\ & \quad = \{(j+1, i+1) : D[i,j] = 0\} \\ & \quad\ \cup \{H[i-1,j] : D[i,j] = D[i-1,j] + w(p[i],\text{-})\} \\ & \quad\ \cup \{H[i,j-1] : D[i,j] = D[i,j-1] + w(\text{-}, s[j])\} \\ & \quad\ \cup \{H[i-1,j-1] : D[i,j] = D[i-1,j-1] + w(p[i], s[j])\}. \end{aligned}$$

Conditions CS3 and CS3M can be checked by doing the same thing for the reverse stings of p and s. Let p^r and s^r be the reverse strings p and s, respectively. Let D^r and H^r for p^r and s^r be the correspondences of D and H for p and s. Then, conditions CS3 and CS3M are satisfied for $s[h..j]$ if and only if $H^r[m, n-h+1] = (n-j+1, 1)$.

We use the suffix tree of s for generating all substrings of s as candidate patterns without duplication. *Suffix tree $\mathcal{T}$ of string* $s[1..n]$ is a labeled tree that has n leaves[1]. Each leaf is labeled by different natural number $i \in \{1, 2, ..., n\}$, and the path from the root node to the leaf labeled i corresponds to suffix $s[i..n]$. Each edge (v, u) also has label $(f, t) \in \{1, 2, ..., n\} \times \{1, 2, ..., n\}$ which represents string $s[f..t]$. Each suffix $s[i..n]$ is the concatenation of the sequence of strings represented by edge labels on the path from the root node to the leaf node labeled i. It is well known that the suffix tree of a string $s[1..n]$ can be constructed in $O(n)$ time [7].

Example 3 *The suffix tree for the string s in Example 1 is shown below.*



Our algorithm, named *EnumSubstrFLOO* (Enumerate Substrings with Frequent Locally Optimal Occurrences), is shown in Figure 1. The algorithm is composed of two stages, forward stage (lines 1-4) and backward stage (lines 5-8). In forward stage, all the occurrences that satisfy conditions CS1, CS2 and CS2M are listed for each candidate pattern string. In backward stage, in which the given string and candidate patterns are reversed, all the occurrences that satisfy conditions CS1, CS3 and CS3M are counted for each candidate pattern string if it also belongs to the list made in the forward stage. When the count is at least the minimum support σ, the pattern is printed out (line 10 in algorithm Optimal_Check in Figure 1).

Our algorithm generates all the substrings of s (or its reversed string s^r) by depth first traversal of the suffix tree of s (or s^r). The traversal is done by algorithm Optimality_Check. Given node v in a suffix tree, for each its child node u, the algorithm generates all the substrings of s (or s^r) that are concatenations of the substring corresponding to the path from the root node to v and non-empty prefixes of the label string of the edge (v, u) as candidate patterns ($s[f-l..j]$ at lines 6 and 9) and calls itself recursively for u.

If patterns' prefixes with length l are the same, then $D[i, j]$ and $H[i, j]$ are the same for all $0 \leq l, 0 \leq j \leq n$. Thus, we can use D and H calculated at a parent node in the calculation for a child node. In the calculation of $D[m, j]$ and $H[m, j]$,

[1]Precisely speaking, there are $n+1$ leaves including the end mark.

the algorithm keeps only the last rows of table D and H, which is a popular technique to enhance space efficiency for alignment algorithms. Also note that, in algorithm CalDH in Figure 2, the value (h, g) of $H[m, j]$ is not calculated correctly but always set to $(h, 2)$ when $g > 1$ because the value g is not necessary in such a case. (See lines 1 and 4 in CalDH.)

Let us consider the time complexity of EnumSubstrFLOO. First, CalDH trivially runs in $O(n)$ time. Both Forward_Check and Backward_Check run in $O(n)$ time because not only CalDH but also other parts of the algorithm runs in $O(n)$ time. Forward_Check and Backward_Check are called at most $O(n^2)$ times because there are n leaves and they are called at most n times for the path from the root to each leaf. Thus, total computational time of them is $O(n^3)$. The inside of the for-loop beginning at line 1 and ending at line 15 in algorithm Optimality_Check is executed $O(n)$ times in total, and the inside of the for-loop beginning at line 4 and ending at line 13 is executed $O(n^2)$ times in total by the above reason, so $O(n^3)$ time is needed for all the executions of Optimal_Check without the time needed for executions of Forward_Check and Backward_Check. In algorithm EnumSubstrFLOO, other parts needs only $O(n)$ time, so it runs in $O(n^3)$ time totally. The space complexity is $O(n^3)$ because $O(n^2)$ occurrence lists are saved and each of them is $O(n)$.

Theorem 1 *Algorithm EnumSubstrFLOO runs in $O(n^3)$ time and (n^3) space.*

Since frequent patterns do not have anti-monotone property, pruning of the search space using the property cannot be applied. However, maximally possible frequency is the number of $H[j]$ with the second component $g = 1$, which can be used for pruning the search space. In our experiments, we used an algorithm using such pruning.

Remark 1 *Algorithm EnumSubstrFLOO is separated into two stages and all the candidate occurrences must be kept at the beginning of the second(backward) stage. This is why $O(n^3)$ space is needed instead of $O(n^2)$ space. Actually, we can achieve $O(n^2)$ space calculation by executing Backward_Check right after each execution of Forward_Check. However, time complexity becomes $O(n^4)$ by this modification because D^r and H^r must be calculated from the beginning in every execution of Backward_Check.*

3 Mining Approximate Forests

3.1 Problem Setting

Let T be a labeled ordered tree whose nodes are labeled a letter in an alphabet Σ. Let n denote the number of nodes in T. The id of node v in T is defined as d if v is visited dth in the postorder traversal of T. We call the node with id d as node d.

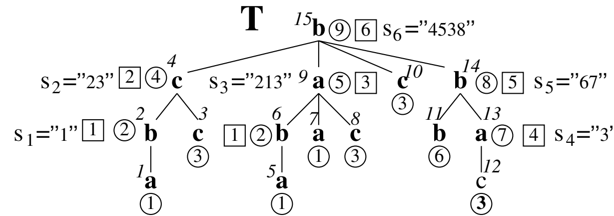
The *(bottom-up) subtree* of T rooted by node d, which is denoted by T_d, is the subtree that is composed of all the descendants of node d. *Child, ancestor and descendant subtrees* of subtree T_d are subtrees rooted by child, ancestor and descendant

nodes of node d, respectively. Subtrees rooted by nodes whose parent is the same node is said to be *sibling subtrees*. A subtree rooted by an internal node is called an *internal subtree*. We regard all isomorphic subtrees as the same subtrees. Let $\rho(T)$ denote the number of distinct subtrees in T, and let $\rho_1(T)$ denote the number of distinct internal subtrees. We define the id of each distinct (internal) subtree as q if the subtree is the qth-appeared distinct (internal) subtree in the sequence of subtrees $T_1, T_2, ..., T_n$, where T_i is the subtree rooted by node i, namely, the subtree rooted by the node visited ith in the postorder traversal of T. We also call the (internal) subtree with id q as (internal) subtree q.

A contiguous sibling subtrees of T is called a *subforest* of T. Every subforest except T itself can be represented by a triple (d, h, j): subforest (d, h, j) of T is a subforest that is composed of a child subtree sequence of T_d from the hth one to the jth one.

Let Σ_T denote the set of subtree ids of T, namely, $\{1, 2, ..., \rho(T)\}$. Any forest that is composed of subtrees in T can be denoted by a string on Σ_T. Let $s_q(= s_q[1..n_q])$ denote the id sequence of the child subtrees of internal subtree q, where n_q denotes the number of child subtrees of internal subtree q. A subforest (d, h, j) is also represented by $s_q[h..j]$ when the internal subtree id of T_d is q. A distinct subforest of T can be uniquely represented by an id sequence of its component subtrees.

Example 4 *Consider a labeled ordered tree T shown below.*



Tree T contains 15 nodes, 9($=\rho(T)$) distinct subtrees, and 6($=\rho_1(T)$) distinct internal subtrees. A number written above each node is its node id, a circled number near each node is the id of the subtree rooted by the node, and a boxed number near each internal node is the id of the internal subtree rooted by the node. Subforest $(9, 1, 2)$ *of T is the forest that is composed of three nodes 5, 6, and 7. For each internal node q, s_q defined above is also shown near q. Subforest* $(9, 1, 2)$ *is also represented by* $s_3[1..2] =$ *"21".*

As a real-valued score function on $(\Sigma_T \cup \{\text{-}\}) \times (\Sigma_T \cup \{\text{-}\})$, we consider a score function w_T based on *tree mapping* [11]. Let T^1 and T^2 be labeled ordered trees whose nodes are labeled a letter in Σ. A *tree mapping* M from T^1 to T^2 is a set of node pairs $M \subseteq V(T^1) \times V(T^2)$ satisfying $u_1 = u_2 \Leftrightarrow v_1 = v_2$ for any $(u_1, v_1), (u_2, v_2) \in M$, where $V(T^i)$ denotes the set of nodes in T^i for $i = 1, 2$. The score of mapping M is defined as

$$\text{score}(M) = \sum_{(u,v)\in M} w(l(u), l(v))$$

$$+ \sum_{u:u \in T_1, \{v:(u,v) \in M\} = \emptyset} w(l(u), \text{-})$$
$$+ \sum_{v:v \in T_2, \{u:(u,v) \in M\} = \emptyset} w(\text{-}, l(v)),$$

where $l(u)$ and $l(v)$ are labels of nodes u and v, and w is a score function on $(\Sigma \cup \{\text{-}\}) \times (\Sigma \cup \{\text{-}\})$. Let $\mathcal{M}^{\mathrm{MC}}(x, y)$ denote a set of tree mapping from subtree x of T (or an empty tree '-') to subtree y of T (or an empty tree '-') satisfying a certain condition MC. Then, a score function w_T on $(\Sigma_T \cup \{\text{-}\}) \times (\Sigma_T \cup \{\text{-}\})$ is defined as

$$w_T(x, y) = \max_{M \in \mathcal{M}^{\mathrm{MC}}(x,y)} \mathrm{score}(M).$$

As a condition MC, we adopt the condition that M must be a *constrained Tai mapping* [13]. Tree Mapping is said to be a constrained Tai mapping if, for any $(u_1, v_1), (u_2, v_2), (u_3, v_3) \in M$, the following conditions are satisfied:

MC1 M must be *Tai mapping* [11], that is,

(1) u_2 is a proper ancestor of $u_1 \Leftrightarrow v_2$ is a proper ancestor of v_1,

(2) u_2 is placed at the left of u_1 in the sibling order $\Leftrightarrow v_2$ is placed at the left of v_1 in the sibling order, and

MC2 the least common ancestor of u_1 and u_2 is a proper ancestor of $u_3 \Leftrightarrow$ the least common ancestor of v_1 and v_2 is a proper ancestor of v_3.

Note that the score function using Tai mapping condition only is known to correspond to general tree edit distance. However, the time complexity of the best known algorithm calculating that score function is $O(n^3)$. On the other hand, the score function using constrained Tai mapping condition can be calculated in $O(n^2)$ time [13]. Furthermore, condition MC2 seems a reasonable constraint as a structure correspondence.

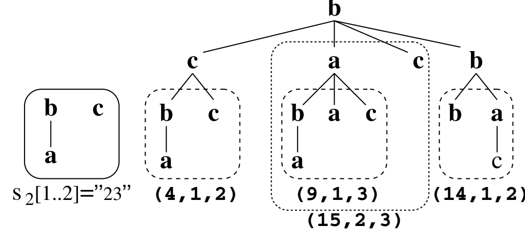
We extend locally optimal occurrences of a pattern string to those of a pattern forest represented by a string $p[1..m]$ on Σ_T.

Definition 3 *Let $p[1..m]$ a string on Σ_T. Subforest (d, h, j) of T is said to be a* locally optimal occurrence *of forest $p[1..m]$ if the following conditions are satisfied for the internal subtree id q of T_d:*

CT1 *(left-right optimality) substring $s_q[h..j]$ of s_q is a locally optimal occurrence of string $p[1..m]$.*

CT2 *(upward-downward optimality)*
$S_{OPT}(p[1..m], s_q[h..j]) \geq S_{OPT}(p[1..m], s_{q'}[h'..j'])$ for all $q' (\neq q)$ that is the internal subtree id of an ancestor or descendant subtree of T_d and for all $1 \leq h' \leq n_{q'} + 1, h' - 1 \leq j' \leq n_{q'}$.

Example 5 *Let T be a labeled ordered tree considered in Example 4. Let score function w on $(\Sigma\cup\{\text{-}\})\times(\Sigma\cup\{\text{-}\})$ be the same one defined in Example 1. Then, the locally optimal occurrences of forest "23" in T are subforests $(4,1,2)$, $(9,1,3)$ and $(14,1,2)$. (See the figure below.)*



Note that substring $s_6[2..3]$ of s_6 is a locally optimal occurrence of string "23" but subforest $(15,2,3)$ is not a locally optimal occurrence of forest "23" because $3 = S_{OPT}(\text{"23"}, s_6[2..3]) < S_{OPT}(\text{"23"}, s_3[1..3]) = 5$ which means that CT2 does not hold for subforest $(15,2,3)$. Subforest $(15,2,3)$ includes subforest $(9,1,3)$ and only one of them is a locally optimal occurrence of forest "23" by virtue of the introduction of upward-downward optimality.

We also extend minimality of a locally optimal occurrence defined for a string to that for a tree. Condition CT2M guarantees that no minimal locally optimal occurrence contains other such occurrences.

Definition 4 *Let $p[1..m]$ a string on Σ_T. Subforest (d,h,j) of T is said to be a* minimal *locally optimal occurrence of forest $p[1..m]$ if (d,h,j) is a locally optimal occurrence of $p[1..m]$ and satisfies the following minimality conditions for the internal subtree id q of T_d:*

CT1M *substring $s_q[h..j]$ of string s_q is a minimal locally optimal occurrence of string $p[1..m]$*

CT2M $S_{OPT}(p[1..m], s_q[h..j]) > S_{OPT}(p[1..m], s_{q'}[h'..j'])$ *for all $q'(\neq q)$ that is the internal subtree id of a descendant subtree of T_d and for all $1 \leq h' \leq n_{q'}+1, h'-1 \leq j' \leq n_{q'}$.*

Problem 2 *Given a labeled ordered tree T, a score function w and a natural number σ, enumerate distinct subforests $s_q[h,j]$ of T such that the number of minimal locally optimal occurrences of $s_q[h..j]$ in T is at least σ.*

Example 6 *Let a labeled ordered tree T and a score function w be those considered in Example 5. When $\sigma = 3$, the solution of Problem 2 is the set of subforests $s_1[1]$, $s_5[1]$, $s_2[2]$, $s_2[1]$, $s_2[1..2]$, $s_5[1..2]$, namely, three height-0 subtrees composed of the node labeled "a", "b" or "c", one height-1 subtree composed of two nodes labeled "b" and "a", and two height-1 subforests composed of three nodes labeled "a", "b" and "c".*

EnumSubforestFLOO(T)
Input: T: labeled ordered tree having n nodes

1: Calculate the subtree id and internal subtree id for each subtree of T
2: For all $q = 1, 2, ..., \rho_1(T)$,

$$N[q] \leftarrow \text{the number of internal subtrees } q \text{ in } T$$
$$I[q][k] \leftarrow \text{the root node id of the } k\text{th occurrence of internal subtree } q \text{ for } k = 1, 2, ..., N[q]$$

3: For all $0 \le q \le r \le \rho(T)$,
$w_T(q, r) \leftarrow$ tree edit distance between subtree q and r.
4: For all $q = 1, 2, ..., \rho_1(T)$,
$s_q \leftarrow$ subtree-id seq. of child subtrees of internal subtree q.
5: Make the generalized suffix tree $\mathcal{T}$
for the set of strings $\{s_q : q = 1, 2, ..., \rho_1(T)\}$.
6: **for** all $1 \le q \le \rho_1(T), 0 \le j \le n_q$ **do**
7: $(D_q[j], H_q[j]) \leftarrow (0, (j + 1, 1))$
8: **end for**
9: $v \leftarrow$ the root node of $\mathcal{T}$
10: Optimality_Check_T($\{s_q\}, v, 0, \{D_q\}, \{H_q\}$,'F')
11: $s^r_q \leftarrow$ reversed string of $s_{\rho_1(T)-q+1}$ for $q = 1, 2, ..., \rho_1(T)$
12: Make the generalized suffix tree $\mathcal{T}^r$
for the set of reversed strings $\{s^r_q : q = 1, 2, ..., \rho_1(T)\}$.
13: **for** all $1 \le q \le \rho_1(T), 0 \le j \le n_q$ **do**
14: $(D_q[j], H_q[j]) \leftarrow (0, (j + 1, 1))$
15: **end for**
16: $v^r \leftarrow$ the root node of $\mathcal{T}^r$
17: Optimality_Check_T($\{s^r_q\}, v^r, 0, \{D_q\}, \{H_q\}$,'B')

Optimality_Check_T($\{s_q\}, v, l, \{D^0_q\}, \{H^0_q\}$,mode)
Input: $\{s_q\}$: set of (reversed) strings
v: a node of the generalized suffix tree of $\{s_q\}$
l: the depth of node v
$\{D^0_q\}$: $D^0_q[j] = D_q[l, j]$
$\{H^0_q\}$: $H^0_q[j] = H_q[l, j]$

1: **for** each child node u of node v **do**
2: $(q_0, (f, t)) \leftarrow$ label of edge (v, u)
3: **for** all $1 \le q \le \rho_1(T), 0 \le j \le n_q$ **do**
4: $(D_q[j], H_q[j]) \leftarrow (D^0_q[j], H^0_q[j])$
5: **end for**
6: **for** $j = f$ to t **do**
7: **if** mode=F **then**
8: Forward_Check_T($\{s_q\}, s_{q_0}[f - l..j], \{D_q\}, \{H_q\}$)
9: **else**
10: occ_list$\leftarrow$ list stored at the position j of edge (v, u)
11: **if** Backward_Check_T($\{s_q\}, s_{q_0}[f - l..j], \{D_q\}, \{H_q\}$, occ_list) $\ge \sigma$ **then**
12: Print $s_{q_0}[n-j+1..n-(f-l)+1]$ as one of the answers
13: **end if**
14: **end if**
15: **end for**
16: Optimality_Check_T($\{s_q\}, u, l+t-f+1, \{D_q\}, \{H_q\}$,mode)
17: **end for**

Figure 3: Algorithms EnumSubforestFLOO and Optimality_Check_T

3.2 Algorithm

Since the half of the conditions, CT1 and CT1M, that occurrences must satisfy in Problem 2 are the same as conditions CS1-3 and CS2-3M in Problem 1, an algorithm for Problem 2 can be designed by extending EnumSubstrFLOO. The extended algorithm is called EnumSubforestFLOO shown in Figure 3.

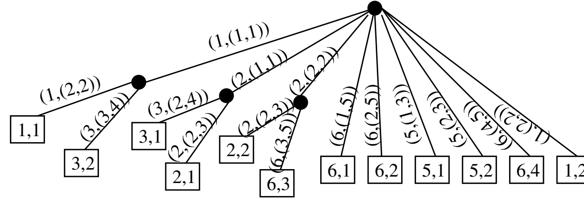
The extensions are mainly the following two points.

E1 For a pattern forest which is represented by a string $p[1..m]$ on Σ_T, the number of locally optimal occurrences of $p[1..m]$ in string s_q are summed up for all internal subtree ids $q = 1, 2, ..., \rho_1(T)$. (Precisely speaking, the number of occurrences in s_q must be multiplied by $N[q]$, where $N[q]$ is the number of internal subtrees q in T.)

E2 Occurrences must also satisfy upward-downward optimality and minimality, namely, CT2 and CT2M.

For extension E1, EnumSubforestFLOO makes the *generalized suffix tree* [7] for the set of strings $\{s_1, s_2, ..., s_{\rho_1(T)}\}$ in order to generate all candidate patterns efficiently without duplication. (See lines 5 and 12 in EnumSubforestFLOO.) In the generalized suffix tree, each edge label also contains the information of which string: edge label $(q_0, (f, t))$ means string $s_{q_0}[f..t]$. (See line 2 in Optimality_Check_T in Figure 3.)

Example 7 *The generalized suffix tree for the set of strings* $\{s_1, s_2, s_3, s_4, s_5, s_6\} =$ {"1", "23", "213", "3", "67", "4538"}, *which is the set of child subtree id sequences of internal subtrees for* T *in Example 4, is shown below. Note that leaf label* q, i *corresponds to* $s_q[i..n_q]$.



For extension E2, downward optimality and minimality, which is described by CT2M alone, are checked in Forward_Check_T and upward optimality of CT2 is checked in Backward_Check_T.

In Forward_Check_T, which is shown in Figure 4, CT2M is checked using array E after $D_q[m, j]$ and $H_q[m, j]$ are calculated for all q and j. The value to be stored in $E[i]$ is the maximum alignment score $S_{\mathrm{OPT}}(p[1..m], s_{q'}[h, j])$ among those for $s_{q'}[h..j]$ such that internal subtree q' is a descendant subtree of internal subtree q (subtree i) and $1 \leq h \leq n_{q'} + 1, 0 \leq j \leq n_{q'}$. Then, CT2M holds for subforest (d, h, j) of any T_d with internal subtree id q if and only if $D_q[m, j] > E[s_q[k]]$ for all $h \leq k \leq j$ and $H_q[m, j] = (1, h)$, which are checked at lines 8 and 9 in Forward_Check_T. The values of $E[i]$ for $i = 1, 2, ..., \rho(T)$ can be calculated in a bottom-up manner by propagating the value for child subtrees of subtree i (line 14) and maximum score for forests composed of those child subtrees (line 12) to subtree i. Note that E-values for child subtrees are calculated before E-values for their parent subtrees in Forward_Check_T because internal subtree ids are in node-id order of their root nodes, and node ids are in visit order by the postorder traversal of T.

Forward_Check_T($\{s_q\}, p, \{D_q\}, \{H_q\}$)
Input: $\{s_q\}$: set of strings
$p[1..m]$: pattern
Update: $\{D_q\}$: $D_q[j] = D_q[m-1, j] \rightarrow D_q[m, j]$
$\{H_q\}$: $H_q[j] = H_q[m-1, j] \rightarrow H_q[m, j]$

1: **for** $q = 1$ to $\rho_1(T)$ **do** CalDH($s_q, p[m], D_q, H_q$)
2: **for** $k = 1$ to $\rho(T)$ **do** $E(k) \leftarrow 0$
3: occ_list←NULL
4: **for** $q = 1$ to $\rho_1(T)$ **do**
5: $i \leftarrow$ subtree id of internal subtree q
6: **for** $j = 1$ to n_q **do**
7: $(h, g) \leftarrow H_q[j]$
8: **if** $D_q[j] > 0$ and $g = 1$ **then**
9: **if** $D_q[j] > E[s_q[k]]$ for all $h \le k \le j$ **then**
10: Add (q, h, j) to the head of occ_list
11: **end if**
12: **if** $D_q[j] > E[i]$ **then** $E[i] \leftarrow D_q[j]$
13: **end if**
14: **if** $E[s_q[j]] > E[i]$ **then** $E[i] \leftarrow E[s_q[j]]$
15: **end for**
16: **end for**
17: Store occ_list at an appropriate position of the edge corresponding to p^r in the generalizes suffix tree $\mathcal{T}^r$

Backward_Check_T($\{s_q^r\}, p^r, \{D_q\}, \{H_q\}$,occ_list)
Input: $\{s_q^r\}$: set of reversed strings
$p^r[1..m]$: reversed pattern
Update: $\{D_q\}$: $D_q[j] = D_q[m-1, j] \rightarrow D_q[m, j]$
$\{H_q\}$: $H_q[j] = H_q[m-1, j] \rightarrow H_q[m, j]$
Output: f: number of minimal locally optimal occurrences of p

1: **for** $q = 1$ to $\rho_1(T)$ **do** CalDH($s_q^r, p^r[m], D_q, H_q$)
2: **if** occ_list= NULL **then** return 0
3: $f \leftarrow 0$
4: $(q_F, h_F, j_F) \leftarrow$ the first element of occ_list
5: **for** $k = 1$ to n **do** $F[k] \leftarrow 0$
6: **for** $q = 1$ to $\rho_1(T)$ **do**
7: $\overline{q} \leftarrow \rho_1(T) - q + 1$
8: **for** $k = 1$ to $N[\overline{q}]$ **do**
9: **for** all child node d of node $I[\overline{q}][k]$ **do**
10: **if** $F[I[\overline{q}][k]] > F[d]$ **then** $F[d] \leftarrow F[I[\overline{q}][k]]$
11: **end for**
12: **end for**
13: **for** $j = 1$ to n_q **do**
14: **if** $q = \rho_1(T) - q_F + 1$ and $j = n_q - h_F + 1$ **then**
15: $(h, g) \leftarrow H_q[j]$
16: **if** $D_q[j] > 0$, $g = 1$ and $h = n_q - j_F + 1$ **then**
17: **for** $k = 1$ to $N[q_F]$ **do**
18: **if** $D_q[j] \ge F[I[q_F][k]]$ **then** $f \leftarrow f + 1$
19: **end for**
20: **end if**
21: **if** no next element in occ_list **then** return f
22: $(q_F, h_F, j_F) \leftarrow$ the next element of occ_list
23: **end if**
24: **if** $D_q[j] > 0$, $g = 1$ **then**
25: **for** $k = 1$ to $N[\overline{q}]$ **do**
26: **for** $d = (n_q - j + 1)$th to $(n_q - h + 1)$th child node id of node $I[\overline{q}][k]$ **do**
27: **if** $D_q[j] > F[d]$ **then** $F[d] \leftarrow D_q[j]$
28: **end for**
29: **end for**
30: **end if**
31: **end for**
32: **end for**

Figure 4: Algorithms Forward_Check_T and Backward_Check_T

In Backward_Check_T, which is also shown in Figure 4, upward optimality of CT2 is checked using array F after $D_q[m, j]$ and $H_q[m, j]$ for reversed strings s_q^r are calculated for all q and j. The value to be stored in $F[d]$ for node d is the maximum alignment

score $S_{\mathrm{OPT}}(p[1..m], s_{q'}[h..j])$ among those for $s_{q'}[h..j]$ such that internal subtree q' is an ancestor subtree of subtree T_d with internal subtree id q. Then, the upward optimality part of CT2 holds for subforest (d,h,j) of subtree T_d with subtree id q if and only if $D_q[m,j] \geq F[d]$ and $H_q[m,j] = (1,h)$, which are checked at line 18 in Backward_Check_T. The value of $F[d]$ for $d = 1, 2, ..., n$ can be calculated in a top-down manner by propagating the value of its parent subtree (lines 8-12) and the maximum score for the substrings of s_q that contains the correspondence of T_d (lines 24-30) to subtree T_d, where q is the internal subtree id of the parent subtree of T_d. Note that F-values for parent subtrees are calculated before F-values for their child subtrees in Backward_Check_T because the order of reversed strings s_q^r is also reversed. (See line 11 in EnumSubforestFLOO.)

Let us consider the complexity of EnumSubforestFLOO. First, consider the preparation phase of EnumSubforestFLOO. Subtree ids can be calculated in $O(n)$ time in average using hashing technique. Internal subtree ids can be calculated in $O(n)$ time from subtree ids. It is easy to check that N and I can be calculated in $O(n)$ time at line 2 in algorithm EnumSubforestFLOO. Score function $w_T(q,r)$ for all $0 \leq q \leq r \leq \sigma(T)$ can be calculated in $O(n^2)$ time when constrained Tai mapping is adopted to define w_T because $w_T(q,r)$ for a pair of subtrees q and r can be calculated in $O(n_q n_r)$ time and $w_T(q,r)$ for all subtree pairs are calculated on the way to calculating $w_T(\sigma(T), \sigma(T))$, where n_q and n_r are number of nodes in subtrees q and r, respectively. All strings s_q can be set in $O(n)$ time because the length sum of all strings is $O(n)$. Unfortunately, Forward_Check_T and Backward_Check_T run in $O(n^2)$ time in worst case because pattern with length $O(n)$ may occur $O(n)$ times. (See line 9 of Forward_Check_T and line 26 of Backward_check_T.) Since complexity of the other parts is the same as that for EnumSubstrFLOO, the total time complexity is $O(n^4)$ in average and the space complexity is (n^3).

Theorem 2 *Algorithm EnumSubforestFLOO runs in $O(n^4)$ time in average and $O(n^3)$ space.*

4 Application to SRR Extraction

In this section, we propose a method using algorithm EnumSubforestFLOO to extract search result records (SRRs) from result pages returned by search engines for submitted keywords.

4.1 Method

Given the tag trees of result pages of a search engine, our method outputs a frequent forest as a wrapper for the search engine. Then, SRR extraction is done by enumerating all minimal locally optimal occurrences of the wrapper forest.

In almost all search sites, SRRs appear as repetitions of almost the same tag structure, namely, as occurrences of a frequent approximate pattern forest. Thus, given an appropriate minimum support σ, an approximate pattern forest for SRRs is contained in the set of frequent forests. However, it also contains approximate pattern forests for the following ones:

1. SRRs with shifted boundaries,
2. SRRs with wrong unit size (connected SRRs),
3. other data records and
4. popular small components.

First, candidate patterns are narrowed by removing patterns consisting of only one node and patterns with overlapping occurrences. Most patterns of 2 and some patterns of 4 above are removed by this operation. Let C denote the narrowed candidate set. Then, each candidate pattern p in C is evaluated from the following three measures:

ave_sz(p) average number of nodes over all occurrences of p;

total_text_len(p) summed text length of all occurrences of p;

ave_text_len(p) average text length over all occurrences of p.

The patterns of 1 above tend to have smaller total_text_len because its frequency is one smaller when SRRs appears contiguously. The patterns of 3 above tend to have smaller ave_size and total_text_len in a search result page, and the patterns of 4 above tend to have smaller ave_text_size. Each of these three measures is normalized by dividing by its maximum value among those of all candidates in C, and weighted combined score

$$\frac{\text{ave_sz}(p)}{\max_{p' \in C} \text{ave_sz}(p')} + \alpha \frac{\text{total_text_len}(p)}{\max_{p' \in C} \text{total_text_len}(p')} + \beta \frac{\text{ave_text_len}(p)}{\max_{p' \in C} \text{ave_text_len}(p')}$$

is calculated for each candidate p, where α and β are parameters to be specified. In our experiments, we set $\alpha = 1.0$ and $\beta = 0.2$. Finally, we select one candidate pattern with the maximum above score.

For performance improvement, the following three heuristics are incorporated into our method.

- For utilizing a no result page when it is available, the score of a candidate pattern that has occurrences in a no result page is discounted by factor γ. We set γ to 0.7 in our experiment.

- In result pages generated by some search engine, all appearances of a search keyword are emphasized using tag <b> or tag <strong>, which causes low similarity of tag structure between two distinct SRRs. So, we added a preprocessing in which every such emphasis tag and its corresponding end tag are removed when non-space text appears between its last preceding tag and itself.
- When the variance on the number of nodes in a subforest corresponding to an SRR is large, pattern forest corresponding to a large SRR might not be so similar to subforests corresponding to small SRRs. In order to find a pattern forest that is similar to all the subforests corresponding to the SRRs in a given tree, we finally select a pattern forest p with the largest total_text_len(p) among those that is similar to the one with the maximum above score.

4.2 Experiments

We conducted experiments on SRR extraction to demonstrate effectiveness of our method proposed in Sec. 4.1.

4.2.1 Methodology

We used data set 1 and data set 2 collected for performance evaluation of ViNTs [14]. Data set 1 and 2 are composed of result pages of 100 and 101 search engines, respectively. The data sets contain 11 pages for each search engine, the six pages of them are for training and the rest five pages are for test. One of the training pages for each search engine is no result page. Here, the set of all the training pages in data set $x(=1,2)$ is called data set x TR, and the set of all the test pages is called data set x TE.

As in the experiments conducted by Zao et al. [14], target SRRs are records of just the major section for each search result page. If more than 25 target SRRs are contained in one result page, just 25 records are counted as target ones[2] to avoid that the overall performance depends too much on the performance for the result pages that contains many records.

We implemented our method using C++ language and ran our programs on Dell Dimension 8200 (Pentium-4 2.20GHz, 512KB cache, 1GB memory) with Ubuntu 8.10.

For each search engine, we first made a wrapper by our proposed method using all the six training pages. Then, we extracted estimated SRRs from each training or test page using the wrapper. Note that no result page contained in the training data set

[2] In [14], they regarded the first 25 records as the target ones, but if do so, the problem of how to calculate precision measure arises. Without decreasing target records, we normalized the number of extracted records and the number of correct records by multiplying them by 25/(the number of target records) in our evaluation.

Table 1: Extraction performance comparison with ViNTs. FLOO stands for our method and parenthesized numbers in ViNTs column are the performance of its NV version.

	Data set 1 TR		Data set 1 TE		Data set 2 TR		Data set 2 TE	
	FLOO	ViNTs	FLOO	ViNTs	FLOO	ViNTs	FLOO	ViNTs
#SRRs	6940	6919	6172	6219	6980	6905	5859	5822
#Extracted	6883.0	6905(6833)	6072.0	6169(6111)	6909.8	6872(6465)	5936.2	5801(5525)
#Correct	6832.0	6901(6722)	6017.0	6164(6001)	6702.0	6740(6283)	5729.2	5673(5390)
Recall(%)	98.4	99.7(97.2)	97.5	99.1(96.5)	96.0	97.6(91.0)	97.8	97.4(92.6)
Precision(%)	99.3	99.9(98.4)	99.1	99.9(98.2)	97.0	98.1(97.2)	96.5	97.8(97.6)
F-measure(%)	98.8	99.8(97.8)	98.3	99.5(97.3)	96.5	97.8(94.0)	97.1	97.6(95.0)

was used only for training and not for the SRR extraction. In wrapper construction, minimum support σ was set to 25. We used the following score function w:

$$w(x,y) = \begin{cases} 2 & (x = y,\ x,y \in \Sigma) \\ -6 & (x \neq y,\ x,y \in \Sigma) \\ -3 & (x = \text{-} \ \text{ or } \ y = \text{-}) \end{cases}$$

Note that the optimal alignment score of two trees defined by this score function is at least 0 if and only if 75% of nodes in the two trees coincide.

We evaluated the extraction performance by recall and precision measures that were calculated by

$$\text{Recall} = \frac{\#\text{Correct}}{\#\text{SRRs}}, \quad \text{Precision} = \frac{\#\text{Correct}}{\#\text{Extracted}}.$$

F-measure, the harmonic mean of the above two measure, is also used for performance evaluation:

$$\text{F-measure} = \frac{2\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall.}}$$

4.2.2 Results

Performance of our method is shown in Table 1 together with performance of ViNTs[3]. The most notable characteristic of ViNTs is use of visual (rendering) information, and its NV version of ViNTs is the one without such information use.

Our method performed worse than the normal version of ViNTs but better than its NV version in terms of F-measure. Considering performance improvement potential of our method by using visual information, this result demonstrates effectiveness of our method as a base method of SRR extraction.

To investigate computational efficiency of our method, we measured running time of wrapper construction and SRR extraction for all data.

The average CPU times are shown below together with some measured values indicating input data complexities.

[3]The small difference on the number of target SRRs between our method and ViNTs is due to the difference on interpretation between us and them because the data sets contain no description about what are target SRRs.

phase	wrap. const.	extraction
#page per run	6	1
ave. #node	3710.4	622.0
ave. #subtree	153.8	91.4
ave. len. of subtree seq.	633.6	269.0
ave. CPU time	0.8861	0.0202

According to the above table, wrapper construction takes less than one second in average and SRR extraction takes about only 20 msec. This result for wrapper construction is surprising because it uses EnumSubforestFLOO as a subprocedure that runs in $O(n^4)$ time for a tag tree with n nodes. See the left figure in Figure 5, which is the log-log scale graph plotting the relation between the total length m of distinct subtree sequence of tree T, namely, $m = \sum_{q=1}^{\rho(T)} |s_q|$, and the wrapper construction time for T. (Precisely speaking, not a single tree but 6 trees are inputted to the algorithm that is extended for such a input.) According to the log-log scale linear regression

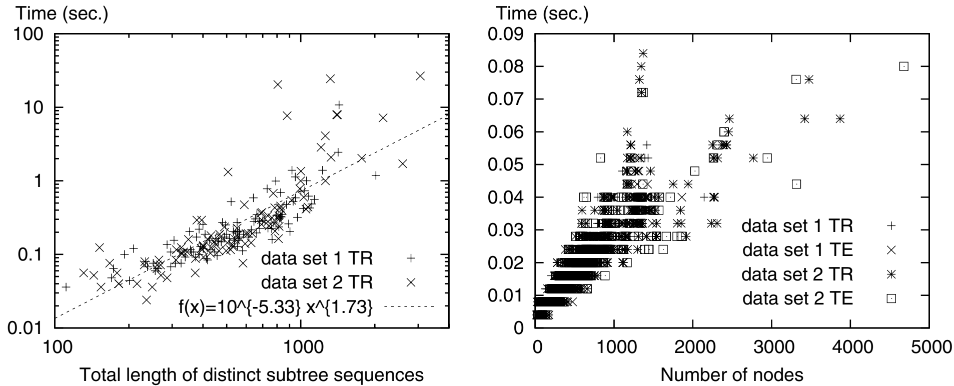


Figure 5: Running times of wrapper construction phase (left) and extraction phase (right)

analysis, wrapper construction takes only $O(m^{1.73})$ time for our data sets. Note that m is at most the number of nodes n and was about $1/6$ of n in average for our training data sets. (See 'ave. len. of subtree seq.' of the above table.) As for SRR extraction, its running time looks linearly depending on the number of nodes according to the right figure in Figure 5.

4.2.3 Discussion

A strong point of using frequent mining to extract SRRs, or data records more generally, is to be able to extract SRRs even if none of them appears consecutively. This could be demonstrated in our experiment using ViNTs data sets. The result pages returned by 'health library' search engine in data set 1 are composed of SRRs in several categories and SRRs in each category are displayed separately following the category name. (See Figure 6.) Our method extracted every SRRs separately without extracting all the SRRs in one category together with its category name as one big

Figure 6: A part of 'care.htm,' which is a result page by 'health library' search engine. The result of three categories are shown in this part.

SRR. Such extraction is basically impossible using methods that detect contiguous occurrences such as MDR [8], ViPER [10] and ViNTs[4] [14].

When the number of SSRs are smaller than given minimum support σ, a wrapper for target SRRs cannot be constructed, which is one weak point of our method. Development of a method that determines σ automatically is desired in the future. In the case that variance on the size of subforests corresponding to SRRs is large, some of them cannot be extracted by the methods using edit score or distance to detect repetitions. Such methods include not only our method but also MDR and ViPER[5]. For this case, methods that detect a record separator like ViNTs seem to work. Taking account of such a separator may improve our method.

4.3 Related Works

Representative methods on automatic data record extraction from web pages are the method in [5], Omini [3], IEPAD [4], MDR [8], ViNTs [14] and ViPER [10]. IEPAD applies string algorithms to a string representation of a web page and the others applies tree algorithms to a tag tree of a web page. IEPAD enumerates maximal

[4]In the experiments on ViNTs, SRRs including a categorical name might have been regarded as correct ones.

[5]ViPER improves this weak point a little by calculating edit score so as to give no penalty to the contiguous repetitions.

frequent substrings as candidate wrapper patterns by means of *exact matching*, and uses multiple string alignment to generalize the patterns. It is NOT fully automatic because the best pattern is selected by users. The other tree-algorithm-based methods are classified into two categories. One is the methods of identifying record boundaries [5, 3, 14]. Omini and the method in [5] first locate the subtree that contains the records of interest and then find a record separator using several heuristics. ViNTs first finds all candidates of a record separator and consecutive similar blocks separated by it, and then selects the best scored separator among them using a heuristically designed score function. The other is the methods of detecting consecutive repetitions [8, 10]. MDR finds repetitions composed of similar forests with the same number of subtrees using tree edit distance to measure the similarity between two subtrees. ViPER finds not only such repetitions but also those composed of similar forests with *variable* number of subtrees using a tree edit distance modified so as to make any repetitions of the same forests match without any additional cost even if the numbers of repetitions are different.

Our method is neither a method of identifying record boundaries nor a method of detecting *consecutive* repetitions. Our approach, which uses frequent mining to find a record structure common among all web pages in a target site, is the same as IEPAD's approach but our method is composed of only one stage to find approximate patterns while IEPAD is composed of two stages, a stage finding exact patterns and a stage generalizing them to approximate ones. Also note that our method is a fully automatic. Our method uses edit distance to obtain approximate patterns like MDR and ViPER but occurrences of a pattern do NOT need to appear consecutively.

According to the previous experimental results [8, 14, 10], ViNTs and ViPER, which use rendering information, are the best performers among all previous systems for SRR extraction task.

5 Concluding Remarks

Main stream of frequent mining on strings or trees is to find patterns that are contained in many strings or trees belonging to a given set of strings or trees. Obtained patterns by such mining can be used as features for classification. In this paper, we proposed a frequent mining appropriate for another use of obtained patterns, namely, estimation of occurrence regions. We believe the importance of such mining and existence of other applications of our mining method.

References

[1] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proceedings of the 11th International Conference on Data Engineering (ICDE 1995)*, pages 3–14, 1995.

[2] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Sakamoto, and S. Arikawa. Efficient substructure discovery from large semi-structured data. In *Proceedings of the 2nd SIAM International Conference on Data Mining (SDM 2002)*, pages 158–174, 2002.

[3] D. Buttler, L. Liu, and C. Pu. A fully automated object extraction system for the world wide web. In *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS'01)*, pages 361–370, 2001.

[4] C. Chang and S. Lui. Iepad: information extraction based on pattern discovery. In *Proceedings of the 10th international conference on World Wide Web*, pages 681–688, 2001.

[5] D. Embley, Y. Jiang, and Y. Ng. Record-boundary discovery in web documents. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, pages 467–478, 1999.

[6] B. W. Erickson and P. H. Sellers. Recognition of patterns in genetic sequences. In D. Sankoff and J. B. Kruskal, editors, *Time Warps, String Edits, and Macromolecules : The Theory and Practice of Sequence Comparison*, chapter 2, pages 55–91. Addison-Wesley, Reading, MA, 1983.

[7] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press, New York, 1997.

[8] B. Liu, R. L. Grossman, and Y. Zhai. Mining data records in web pages. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2003)*, pages 601–606, 2003.

[9] J. P. Schmidt. All hightest scoring paths in weighted grid graphs and their application to finding all approximate repeats in strings. *SIAM Journal on Computing*, 27(4):972–992, 1998.

[10] K. Simon and G. Lausen. Viper: Augmenting automatic information extraction with visual perceptions. In *Proceedings of the 14th conference on Information and Knowledge Management (CIKM 2005)*, pages 381–388, 2005.

[11] K.-C. Tai. The tree-to-tree correction problem. *Journal of the Association for Computing Machinery*, 26(3):422–433, 1979.

[12] M. J. Zaki. Efficiently mining frequent trees in a forest: Algorithms and applications. *IEEE Transaction on Knowledge and Data Engineering*, 17(8):1021–1035, 2005.

[13] K. Zhang. Algorithms for the constrained editing distance between ordered labeled trees and related problems. *Pattern Recognition*, 28(3):463–474, 1995.

[14] H. Zhao, W. Meng, Z. Wu, V. Raghavan, and C. Yu. Fully automatic wrapper generation for search engines. In *Proceedings of the 14th international conference on World Wide Web (WWW 2005)*, pages 66–75, 2005.

[15] F. Zhu, X. Yan, J. Han, and P. S. Yu. Efficient discovery of frequent approximate sequential patterns. In *Proceedings of the 7th IEEE International Conference on Data Mining (ICDM 2007)*, pages 751–756, 2007.